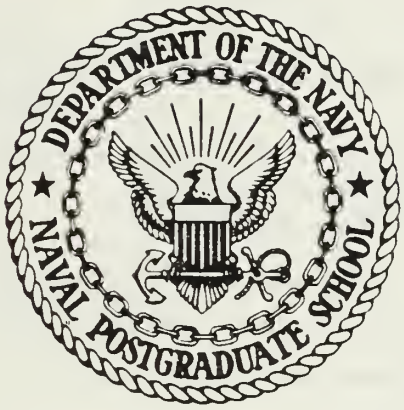
1986

Software portability: a case of the multi-backended database system.

Silberman, Bruce D.

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

SOFTWARE PORTABILITY:
A CASE STUDY OF THE
MULTI-BACKENDED DATABASE SYSTEM

by

Bruce D. Silberman

June 1986

Thesis Advisor: David K. Hsiao

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION val Postgraduate School 6b. OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) nterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification) UNCLASSIFIED
oftware Portability: A Case Study of the Multi-Backended Database System

PERSONAL AUTHOR(S)
Bruce D. Silberman

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
asters Thesis	FROM _____ TO _____	1986 June 20	104

SUPPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	software portability, multi-backended database system, database systems, software systems, network communications

ABSTRACT (Continue on reverse if necessary and identify by block number)

The multi-backended database system (MBDS) is a database system designed for very large databases. MBDS is intended to provide consistent performance with increased capacity (or improved performance at a sustained capacity) by distributing the work of the system among several micro-computers connected to a common communication network. One of the issues central to the MBDS design is the portability of the system's software. (Continued)

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL Prof David K. Hsiao	22b TELEPHONE (Include Area Code) 408-646-2253 22c OFFICE SYMBOL 52Hq

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ABSTRACT (Continued)

This thesis provides a general discussion of the issues involved in software portability, and then presents a case study of the MBDS software system.

Approved for public release, distribution unlimited

Software Portability: A Case Study of the Multi-Backended Database System

by

Bruce D. Silberman
Lieutenant, United States Navy
B. S., Florida Technological University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1986

ABSTRACT

The multi-backended database system (MBDS) is a database system designed for very large databases. MBDS is intended to provide consistent performance with increased capacity (or improved performance at a sustained capacity) by distributing the work of the system among several micro-computers connected to a common communication network. One of the issues central to the MBDS design is the portability of the system's software. This thesis provides a general discussion of the issues involved in software portability, and then presents a case study of the MBDS software system.

TABLE OF CONTENTS

I. AN INTRODUCTION

A. THE BACKGROUND

It should come as no surprise to anyone involved in computer science to say that the essence of the entire field is changing. This is probably true for those persons involved in software development. The trend of rapidly decreasing hardware costs remains steady, and shows no signs of changing course in the near future. Software costs therefore represent an ever increasing percentage of total system costs.

As the level of technology increases, and the relative costs of hardware continues to decrease, the expense of upgrading to larger, more powerful computing systems becomes less restrictive. Except for the ever present (and increasing) cost of new software, the obvious requirement then is for the software to be easily ported from one system to another. Assuming an environment of truly portable software, users would find themselves in a situation where the only real costs to be considered when evaluating a change to a new system would be the cost of the hardware alone.

There are some obvious benefits to this situation. When the need arises for a greater capacity or an increased throughput to meet the new requirements, the upgrade becomes an easier task. Additionally, more powerful software can be purchased at the outset, when the original computing systems are acquired. Since the useful life of a software item would no longer end when its underlying hardware is replaced, higher initial expenses for software could be justified.

Cost is not the only factor in support of software portability. It is often desirable to provide a uniform, consistent environment across several different computing systems. The days of the single mainframe which supports all of the computer needs in an organization are fading. Many organizations now have several different computing systems (usually minis or micros). They may be made by different manufacturers, run by different operating systems, used for different tasks, and are networked together frequently to provide a very powerful computing environment where resources may be shared.

It is also common to see that the same operating system is running on several different computers (e.g., UNIX, which runs on VAX main frames, ISI 32-bit micro-computers, IBM 16-bit micro-computers, et. al.). Conversely, hardware manufacturers often provide a choice of operating systems which will run on the same hardware: and applications developers will provide versions of their software which run under different operating systems, and allow exchange of data between them without modification. In such an environment it is not unreasonable for a user to desire programs which can be moved between several computing systems.

B. AN OVERVIEW

Portability is essentially the ability of software to migrate among different hardware configurations and operating systems with little, or no, modification. This thesis will provide a discussion of techniques which can be used to achieve a higher degree of portability for software in general, and database management software in particular.

The need for portable software is evident through all areas of computer science. database management systems (DBMS) present a particularly strong need for it. Conceptually, DBMS software can be thought of as a specialized operating

system. Like an operating system. DBMS accepts commands from a command language (data language). and uses the commands to manage and manipulate files (data). System users utilize both DBMS and the operating system as support tools. Like operating systems, the DBMS software tends to be complex and expensive. often taking many years to develop.

The DBMS software however presents something of a paradox in that the more powerful DBMS is, the quicker it exceeds its usefulness. This happens not because DBMS ceases to be adequate, but because the hardware on which the system is running can no longer handle all of the work which the DBMS software is trying to accomplish.

The better DBMS is, the more users there are. Unfortunately, the increasing strain which a growing database places on a computing system is not linear. Response times increase drastically, and soon. running nondatabase operations concurrently with DBMS becomes infeasible. Eventually, even running DBMS as a standalone system becomes uneconomical.

One obvious answer to this dilemma is to go to larger. stand alone. computing systems for handling database functions. There is no reason to believe, however, that this is anything but a short-term solution. Databases will almost always continue to grow, so the emphasis is once again on the need for the portable DBMS software which can be moved from one system to another system.

Researchers in the area of database management are finding new ways to improve the efficiency of the DBMS software, and reduce the strain that the DBMS software imposes upon the mainframes called hosts. One current project, is the multi-backended database system (MBDS). MBDS is aiming at meeting both of these goals by placing the DBMS software on independent microcomputers known as the backends which are connected to each other, and to their

host via a communications link. Since most processing with MBDS is on independent computers, no real strain is placed upon the host computer. MBDS provides for increased capacity without decreased performance (or increased performance at the same capacity), by the addition of micro-computers.

MBDS places no restraint on either the host computer type, or the micro-computer backend. There is also no restriction on the method used to establish the communications link. In order to provide this freedom, and since the system is designed with the intention of migrating to various hardware configurations (possibly at frequent intervals), the MBDS software must be easily portable.

Due to the great desire for portability in database software and the additional efforts by the designers of MBDS to achieve it, MBDS represents an excellent example of methods which can be used to provide software portability. The methods used are applicable to the software portability in general, and to the portability of the DBMS software in particular. As such, one implementation of the MBDS design is used for discussion throughout this thesis. A complete description of MBDS design can be found in Reference 1 and Reference 2.

C. THE ORGANIZATION OF THESIS

Chapter II presents a paradigm for a software system, and describes the parts of a computing environment. A discussion is presented on how the various components of a software system (presented in the paradigm) interact with each part of the computing environment. Also discussed are the effects that changing each part of the computing environment have on the components of a software system.

In chapter III we describe the components of the multi-backended database system (MBDS) according to the framework of our paradigm. The porting of MBDS from one computing system to another is discussed, and the effects of the

porting on the MBDS software components are presented. The actual modifications which are necessary to port the MBDS software are presented in chapter IV.

II. THE SOFTWARE-SYSTEM PORTABILITY

Before progressing with any discussion of how to achieve the portability of a software system we must have some concept of what portability is, and what comprises a software system. Only then can we correctly determine what the issues are and their impact upon the portability, and how these issues should be approached and analyzed. Probably no two definitions of the software portability exactly coincide, depending on the background of the person providing the definition, and the scope of the problem at hand. However, there are some general points that all of the different definitions are likely to agree on.

The software-system portability is the ability of a software system to migrate from one to another computing environment (i.e., hardware and operating system combinations) with little, or no, changes of the software system required. Changes which are required should be consistent and of a regular nature, so that these changes can be accommodated, if possible, by some mechanical means.

Given this definition, we now must focus on what constitutes a software system. Software systems can be designed for many different purposes. In order to carry out their intended job they usually have various parts (some times hundreds of them), each of which satisfies some specific requirement. There are usually sections for carrying out normal processing and for creating and manipulating data structures. Most software systems also require access to device drivers to enable terminal I/O or communication with other software systems, either within the same computer or over some distributed network.

Regardless of the software system in question, its portability is not determined by what the software system is intended to do, or how many parts it has, but by how the software system was structured. Therefore, the issues affecting the portability are general to all types of software systems, since all of the software systems have some structure. The concepts to be discussed do not vary, nor does its basis, on whether the software system is a database system, operating system, a compiler, or any other type of systems. In the rest of this chapter we examine what those issues are. A case study on the portability is presented in the next chapter, using the multi-backended database system (MBDS) as an example.

A. A PARADIGM FOR A SOFTWARE SYSTEM

In this section we present a paradigm for the structure and form of a software system. This paradigm is based on the two types of components that are used to construct a software system. They are *system source code* and *operating system commands*. There are three types of system source code, namely,

- machine code,
- assembler code, and
- high-level code.

Operating system commands take the form of

- basic commands,
- command files, and
- utilities.

First, we investigate the system-source-code component. For each type of code in the system-source-code component, our paradigm provides up to three types of processing:

- basic processing,
- runtime-environment processing, and
- translator-environment processing.

In the following discussion, we investigate the form that each of the three kinds of processing takes for each of the three types of the system source code.

The machine code is just binary code written for the particular machine the program runs on. The basic processing functions for machine code include operations such as add, subtract and store. There is no runtime-environment processing, or translator-environment processing available with the machine code.

The assembler code, while it is at a higher level than the machine code, usually results in only a few machine instructions per line of the source code. The capabilities at the basic processing level for the assembler code parallel those of the machine code. However, the assembler code allows logical names and performs the automatic calculation of machine addresses. The runtime-environment processing for the assembler code involves calls to the operating system for performing such functions as reading characters from files, or obtaining the current date and time. Assemblers generally do not provide any translator-environment processing capabilities.

The high-level code refers to compiler languages such as C or Pascal. For the high-level code, basic processing capabilities include mathematical operations, logical comparisons, and assignment of values to logical variables. They also include all those statements in the language which we call the runtime-environment processing or translator-environment processing. The runtime-environment-processing capabilities include operations such as reading and writing files, dynamically allocating the memory and providing calls for communicating with other software systems. The high-level calls to be carried out by the operating

system constitute the runtime-environment processing. The translator-environment processing is any function that is carried out by library routines which are provided as a part of the translator and incorporated into the software system at the link time. Included in the translator-environment processing are operations such as floating-point arithmetic, computation of trigonometric values and manipulation of character strings.

Now let us consider the operating-system-command component of a software system. The basic operating system commands include those for compiling, assembling and linking the source code of all types (i.e., machine, assembler and high-level); and for deleting, copying and renaming files. Command files are lists of basic processing commands which automatically execute, in sequence, or according to some programming logic dictated by the operating system. The utilities of the operating system are prewritten command files which are included as a part of the operating system. Utilities may perform logically complex operations and are designed to aid in the implementation and management of large software systems. Included in these are system libraries and implementation aids such as version control systems and file-creation utilities.

We should take the time here to note an important point. While all of the code is eventually translated into the machine code and runtime libraries may make calls to the operating system, neither of these directly affects the portability of the software. The issues are what types of the system source code that the software system is written in, and which types of processing are performed. These are what determine the portability of the software system.

B. TYPES OF PORTING

We have defined the porting as the movement of a software system form one computing environment to another. There are three parts which make up a computing environment,

- the hardware,
- the operating system, and
- the translator (in the case of the assembled or high-level code).

A software system may be ported by changing any of the three parts of the computing environment. Each of these changes has a different impact on the software system being ported. There may of course be changes to several items at once. For example, a change of operating systems usually requires a change of translators as well, since most compilers are written to run on a particular operating system. We now use the paradigm we have created to examine the effects that each type of porting has on a software system.

1. A Change of the Hardware

For any software system, the most profound effects of changing hardware are in the system-source-code component. Any portions of a program written in the machine code obviously are affected by a change of the hardware. Unless the new hardware used allows for the emulation of the original machine, all machine code must be rewritten. The assembler code usually must be redone as well because it is so closely tied to the structure of the underlying machine.

The high-level code usually does not have to be redone due solely to a change of hardware as long as that new hardware supports a compatible version of the original operating system and that a compiler is available for the high-level language in question. However, this is not always the case. If the high-level

language is not widely used (e.g., BLISS), then there might not be a compiler available with a code generator for the new machine.

Operating-system commands are not affected by a change of the hardware alone, since two versions of an operating system running on two different sets of hardware can usually be expected to provide the same functions. Thus, in the case of a hardware-only change, the type of code used (machine, assembler, or high-level) is what determines the changes to be made to the system source code. There is usually no affect on the operating-system-command component of the software system.

2. A Change of Translators

Changing the translator only has an effect on the assembled and high-level code in the system-source-code component (the machine code does not use a translator). The kinds of processing which the code (assembled or high-level) performs affect the changes which have to be made. Necessary changes to the code which accomplishes the basic processing are determined by the degree of the language's standardization.

In addition to intentional design differences there is a question of whether a particular translator correctly implements the intended language. Formally validating compilers for modern high-level languages is not possible. Most commercial compilers are subjected only to some series of empirical tests. If minor differences from the defined language are detected, manufacturers may decide to document the deviations, rather than pay the expense of trying to correct them. These deviations may affect how well a software system reacts to a new compiler.

Even if the syntax of a language is standardized, changes may be required in portions of the code which do the translator-environment processing.

The runtime libraries provided by the compiler suppliers are not always the same. A good example of this is Pascal's string manipulation routines. While the language itself does not define string types, almost all implementors include them as an extension to the language. However, the format used by each is not standardized and often differs among compilers. While the old and new translators may both provide functions that have the same name and take the same parameters, there may be logical differences in how each of them implements these functions. If these functions perform critical operations in the software system, it may be desirable to avoid the runtime-environment processing whenever possible and write your own routines instead.

For the system-source-code component of a software system, the changes necessitated by a change of translators is determined by the type of processing done by the code. There is usually no effect on the operating-system-command component of the software system.

3. A Change of Operating Systems

It is the operating system which provides the major functionality in a computing environment. A change of the operating system therefore is the most drastic kind of porting. Porting software to a new operating system may affect the assembler-code and high-level-code portions of the system-source-code component. The machine code (which relies only on the underlying machine) does not have to change. The most direct effects of changing operating systems are to those code sections which do the runtime-environment processing. If the new operating system does not provide equivalent functions for those operations, it may become necessary to redesign the logic of entire sections of the code.

The portability of a software system may also be complicated by the fact that a change of operating systems requires some changes to the translator which

the software system has used. This subjects the software system to all of the possible changes it would have to undergo for any other change of translators. Therefore, changing operating systems has an indirect effect on the code which does the translator-environment processing and, possibly, the basic processing as well.

Changing operating systems obviously affects all parts of the operating-system-command component of a software system. Basic commands (e.g., compile, link, search, etc.) are almost always supported in some form by the new operating system; command files usually are available as well. Therefore, the changes required by these commands are a matter of substituting the new formats for the old. The greatest difficulties arise if operating-system utilities have been utilized and the equivalent utilities do not exist in all operating systems. When they do exist, their formats may not be at all similar. In either case, an extensive restructuring may be necessary to maintain the software.

The porting to a new operating system may require major changes to both components of a software system. In the system-source-code component, the assembler and high-level code may have to be changed. These changes may affect the code which does any of the three types of processing. For the operating-system-command component, a major rewriting is usually necessary for all three forms of operating-system commands.

III. THE IMPACT OF THE MBDS DESIGN ON PORTABILITY

In this chapter we present a case study of the porting of a particular software system from one computing environment to another computing environment. Our description of the software system that was ported follows the structure of the paradigm presented in the previous chapter. The description of the actual porting also follows the format we presented in the previous chapter (i.e., describing each part of the computing environment which changed and how the change affected each component of the software system that was ported).

The software system that was ported for our case study is the multi-backended database system (MBDS). MBDS is a database system (for very large databases) that was developed for research purposes at the U. S. Naval Postgraduate School. The basic premise of MBDS is to distribute the work of the database system across several different micro-computers. MBDS uses one computer as the system *controller*, and several other computers (at present, eight) as *backends* to accomplish the majority of the required database-manipulations. There are two reasons why MBDS was chosen for our case study: 1) MBDS has been ported successfully twice since its initial implementation, and 2) the MBDS designers have established portability as a high priority since the initial inception of the MBDS project. Including the first implementation, MBDS has been run in three different computing-environments. Among the three computing-environments, there have been four different hardware configurations (MBDS uses more than one computer per environment), and five different operating systems.

To attain a portable software system, the designers of MBDS strived to develop the MBDS software so that it contained a high degree of hardware and operating-system independence. By engineering MBDS with a minimum amount of dependencies, they greatly enhance the probability that the software system would be easily transportable to new, and varying, hardware/operating-system configurations.

To develop a highly portable database system, the designers of MBDS first identified which portions of the database system software are dependent on either the hardware and/or the operating system. They identified two classes of database system software which are dependent, namely, *communications* software and *disk input/output* software. Communications software is used by the database system to communicate among different computers and to communicate within a computer, referred to as *inter-computer* and *intra-computer communications*, respectively. The communications software is often affected by a change of the operating system (since communications protocols are operating-system dependent) and is also affected by a change of the hardware (since specialized communications drivers are hardware dependent). The disk input/output software is used by the database system to access and process information from the secondary storage. The disk input/output software is also affected by a change of the operating system (since it is operating-system dependent) or by a change of the hardware (since it is dependent on specialized disk drivers).

In general, there is no way to avoid a certain amount of hardware and operating-system dependencies in a database system. Instead, MBDS was developed with techniques which can minimize the effect of changes. There are two distinct approaches to accomplishing this task. First, MBDS uses the concepts of *abstraction* and *encapsulation* to isolate the dependencies of the

communications and disk input/output software. The database system software makes calls to these high-level routines that are dependent on the programming language used in the software system, when there is a need to access the system-dependent software. These calls are generic, e.g., send[message, destination], receive[message, sender], do_disk_io[data, device]. They represent abstractions of the actual functions. The routines themselves (i.e., send, receive, and do_disk_io) are used to encapsulate the system-dependent software. Second, the designers used the concept of a *virtual interface* to develop independent software for communications and disk input/output. The aim of the virtual interface is to utilize abstractions provided by the compiler to accomplish a particular task. These abstractions are usually in the form of library routines for the programming language. As these library routines are supported by different compilers under different operating systems, it is easy to transport the virtual interface from one operating system to another.

MBDS has utilized the abstraction and encapsulation concepts, as well as the creation of a virtual interface. Abstractions and encapsulations are used by the MBDS communications software to provide high-level calls to send and receive messages both among and within computers. Since MBDS is a message-oriented system, all of the inter-computer and the intra-computer communications are accommodated by the abstractions and encapsulations. These techniques are also used by the MBDS disk input/output software to provide a high-level interface for reading (writing) information from (to) the secondary storage. In addition, the MBDS designers also created a virtual interface for disk input/output software. The virtual interface depends on the programming language constructs, and is used to provide a high-level, operating-system-independent paradigm for performing disk input/output via text files.

In the next section we take a closer look at the construction of MBDS according to our paradigm of a software system. An in-depth description of the MBDS design can be found in Reference 1 and Reference 2, descriptions of the implementation of MBDS can be found in Reference 3 and Reference 4.

A. THE MBDS COMPONENTS

The system-software component of MBDS contains only high-level code (there are no sections of machine code or assembler code). This has greatly enhanced the portability of the MBDS software system. The MBDS high-level code (written in the C programming language) performs all three types of processing contained in our paradigm (i.e., basic, runtime-environment and translator-environment processing).

The basic-processing operations of MBDS are the normal C language constructs which would be found in any C program (e.g., for-loops, while-loops, if-then-else statments, etc.). These basic-processing operations carry out the majority of the processing in the MBDS software. We have stated in the previous chapter that portions of the system-source-code component which are written in high-level code and perform basic-processing operations are inherently portable. Therefore, the majority of the MBDS software system is inherently portable. The portions of MBDS which do runtime-environment-processing are those sections of high-level code which perform inter-computer communications, intra-computer communications, disk input/output and system-timer operations.

Translator-environment-processing operations in MBDS are performed by function libraries provided with the C compiler. They consist essentially of routines for performing terminal input/output and routines for the extensive character-string manipulations which the MBDS software performs. MBDS also

performs some disk input/output at this level. when dealing with files used for execution-trace information and user input. All of the manipulation of the database-files is done at the runtime-environment processing level. under the guise of abstraction, encapsulation and virtual interface techniques.

The system-software component of MBDS consists of over eighteen-thousand lines of C code. In order to implement and manage such a large software system the designers of MBDS make extensive use of the operating-system-command component of MBDS. Operating system command-files are used to gather the required system-source-code files in one place, compile and link them into executable files, relocate all executable files in one area for later execution, and then erase any intermediate (temporary) files which are no longer needed. In addition, since the MBDS software system consists of multiple processes running across several computers (usually five processes per computer), command files are utilized to initially start the system in an orderly fashion.

B. PORTING THE MULTI-BACKENDED DATABASE SYSTEM

This section discusses the actual changes made to the MBDS computing-environment each time MBDS was ported, and the effects that each of those changes had on the MBDS software. Each time MBDS was ported, changes were made to both the hardware and the operating system at the same time (the compiler also changed, because of the new operating system). For the sake of clarity, each of these changes is discussed independently.

1. Changing the MBDS Hardware

MBDS has had three different hardware configurations, the first configuration is the original. The controller for MBDS is a Digital Equipment Corporation (DEC) VAX-11/780. The backends are two DEC PDP-11/44s. For

the database store. each backend utilizes a 67-megabyte RM03 disk drive. Inter-computer communications is accomplished using the point-to-point parallel communications link (PCL). a 0.5-megabit bus. Three PCL's are utilized. one from the controller to the first backend, one from the controller to the second backend and one between the two backends.

In the second configuration the controller for MBDS is a DEC VAX-11/750, the backends for MBDS are eight Integrated Solutions Incorporated (ISI) Motorola 68020-based workstations. For the database store, each backend utilizes a 500-megabyte Control Data Corporation (CDC) winchester-type disk drive. Communications is accomplished using an Ethernet. All computers (both controller and backends) share the same Ethernet.

For the third configuration, the controller for MBDS once again is a DEC VAX-11/780, while the backends are upgraded to DEC MicroVax-IIs The communications bus is also upgraded to a DELNI. with DECNET providing the networking software interface. Each backend has a 71-megabyte DEC winchester-type disk drive.

Since all of the MBDS software system consists of either high-level system-source-code or operating-system-commands. we can expect that no changes would be required to the software solely because of a change of the underlying hardware. That is in fact what happened. In both cases. where MBDS was moved from one piece of computing hardware to another. no changes had to be made to the MBDS system-source-code (as a result of the hardware). There were also no changes required the operating-system-command component of MBDS.

2. Changing the MBDS Source-Code Compiler

A change of compilers was experienced by the MBDS high-level code (by circumstance) each time a change of operating systems was made. In the original

MBDS configuration the DECUS C compiler was used for high-level code which was developed on the PDP-11/44 computer (using the RSX-11/M operating system). For code that was developed on the VAX-11/780 computer (using the VMS 3.7 operating system) the EUNICE C compiler was used (EUNICE emulates a UNIX environment on the VMS operating system). For the second MBDS configuration, the UNIX C compiler was used for high-level code on both the VAX-11/750 (using the UNIX 4.2 BSD operating system) and the ISI workstations (also using UNIX 4.2 BSD). In the third version of MBDS the DEC C compiler on Micro-Vax IIs was used (using the MVMS 4.1 operating system). This same C compiler was used to develop high-level code for the VAX-11/780. Since executable-code generated for the Micro-Vax II computer is *upward compatible* with the VAX-11/780, it was possible to develop high-level code for the Vax-11/780 on the MicroVaxs and then copy across the executable files. This avoided any risks of using yet another C compiler, and avoided using the VAX-11/780 for development purposes (since the VAX-11/780 was used for other purposes in addition to the MBDS research).

Changes to system-source-code (as result of changing compilers) would normally be expected in sections of code which perform translator-environment processing. However this was not the case with the MBDS software, there were no changes required for code which performed translator-environment-processing. This is probably because the origin of the C programming language is closely tied to the UNIX operating system. Therefore, most developers of C compilers (for any operating system) provide compiler libraries which are copies of the original UNIX C compiler library.

There were some changes required in the MBDS high-level code which performed basic-process operations (as a result of the new compiler). These

changes were required when the MBDS software was ported from the second to the third versions (i.e., from the UNIX C compiler to the DEC C compiler). Though the actual changes required were simple (and were needed in less than twenty lines of C code), the time required to determine the cause of the problem was considerable. Both compilers accepted the source code (they just interpreted it differently), so there were no error massages produced at compile time. Indications of the compilers' differences did not appear until the new executable files were run and the new version did not work! Keep in mind that a new compiler was not the only change which had taken place. This, along with the fact that no problems had arisen from previous compiler changes, and that the problems were in sections of code which performed basic-processing (the least probable area), served to create a difficult debugging problem.

The difficulties apparently resulted from differences in the way the two compilers assigned precedences when multiple C language operators appeared in one line of code (the C programming language allows certain *short hand* coding constructs to save typing time). The only changes needed were to remove these short-hand lines of code, and rewrite them in a manner similar to the way they would be written in any other high-level language (e.g., Pascal). From a software engineering point of view, the short-hand coding techniques should have been avoided anyway, since they tend to be very cryptic, making the resulting source code difficult to read and maintain.

3. Changing the MBDS Operating System

Portions of the MBDS software have run on five different operating systems. In the original MBDS configurations the controller used the DEC VMS 3.7 operating system, while the backends used the RSX-11/M operating system. For the second version of MBDS, both the controller and the backends used the UNIX

4.2 BSD operating system. In the third version of MBDS, the VMS 4.2 operating system was used for the controller, with MVMS 4.1 running on each of the backends.

As was stated in the previous chapter, a change of operating systems is the most drastic kind of porting for any software system. The MBDS software changes required as a result of a new operating system accounted for the vast majority of work needed, both in terms of time spent and percent of the MBDS software which had to be modified. While all of the MBDS system-source-code which performed runtime-environment-processing had to be modified, the changes were isolated to just a few source files. This was a result of the abstraction and encapsulation techniques used by the MBDS designers. Therefore, it was easy to predict the majority of the changes which must be made to accomplish the porting.

Some minor changes must be made to sections of the MBDS code which performed system-timing operations. These changes consisted mostly of changing the names of functions which were called (from operating system libraries), and rearranging the order of some parameters to those functions. The majority of the system-source-code changes must be made in the sections of code which perform communications.

For the original version of MBDS three different methods of communication were used depending on whether the need was for inter-computer communication, intra-computer communication in the controller, or intra-computer communication in a backend. Inter-computer communication was accomplished using a point-to-point parallel communications link. In the backends, intra-computer communication (under the RSX-11/M operating system) was accomplished with shared memory techniques. For the controller, intra-computer communications

(under the VMS 3.7 operating system) were performed using VMS mailboxes. When the MBDS software was ported to the UNIX operating system none of these communication techniques existed.

Under the UNIX operating system, all communication (inter-, and intra-computer) was accomplished using UNIX sockets. This meant that all code which performed the communications had to be changed. Due to the virtual-interfaces which MBDS had set up, these sections of code were again isolated to just a few source files. There was also the advantage that the virtual-interfaces could accomplish all three types of communications with just one low-level driver (UNIX sockets). However, when the third MBDS version was implemented it did not use UNIX, so all the communications drivers again had to be changed.

The third version of MBDS used VMS mailboxes (like version one's controller) for intra-computer communication in both the controller and the backends. This meant that the source code which had been used in version one's controller could be used (virtually unchanged) for version three's controller. In addition, it could be used for the backends as well (with only minor changes). Inter-computer communications in the third version of MBDS were accomplished using DECNET. Since this technique had not been used in any previous MBDS versions, entirely new communications drivers must be written. The virtual-interface techniques again isolated the changes to just a few files.

The operating-system-command component of MBDS was rewritten each time MBDS was ported to a new operating system. There are in excess of twenty-five command-files used to manage the implementation of the MBDS software. So, while the required changes were mechanical in nature, they required a great deal of time to accomplish simply because of the number of files which were changed. Even though the controller for version one of MBDS ran on the VMS

operating system (like the controller and backends of version three), the controller software only represents about twenty-five percent of the MBDS software. Therefore, while some of the command files from version one could be used in version three (with minor modifications), there were still a great many command files which were rewritten.

IV. A CLOSER LOOK AT THE MBDS PORT

In this chapter we discuss the steps necessary to transform MBDS version two into MBDS version three. We examine the porting process in a chronological order, essentially presenting a journal of the activities used to move the software. A separate detailed discussion is presented of the changes made to the MBDS communications software (which underwent the greatest changes). First, lets look at the sequence of events used to accomplish the porting.

A. THE SEQUENCE OF EVENTS

Porting the MBDS software from one system to another has entailed several distinct phases. The major milestones in the sequence are as follows:

- transfer the MBDS files (of source-code and operating-system commands) from the old computing system to the new one,
- modify the operating-system-command files used for compiling the MBDS software,
- compile the source-code files and correct any compile-time bugs which appear,
- modify the MBDS intra-computer communications software, and implement an *intermediate* version of MBDS with a controller and one backend, both running on the same computer,
- perform run-time testing on the intermediate system to ensure the actions of the implementation are logically correct (according to the MBDS design),
- modify the MBDS inter-computer communications software to implement the final version (MBDS version three),
- confirm the actions of the final version are logically correct.

While some of these phases overlap to some extent, the porting sequence is clearer if they are considered separately.

The files of source-code and operating-system-commands which comprise MBDS (version two) exist on one of the ISI workstations. The first step in the porting process is to copy all of the files to one of the Micro-Vaxs, where the development of MBDS version three has been accomplished. Since the computing systems for both versions of MBDS are connected to a common local-network, the files are copied using the standard communications utilities available.

Once the are were on the new system, the next step is to convert the operating-system-command files. Since the command files are needed to manage the compilation of the MBDS source-code, their conversion must take place before the compilation of the system begins. Each implementation of MBDS contains six independently executed programs (or processes) for the controller and five independent programs (or processes) for each backend. There is a set of command files for each of the independent programs. One command file in the controller source-code (and one in each backend) may call all of the subordinate command files, and in this way the entire set of MBDS processes can be created from the top level. The total number of command files for the controller and backends (in version two) is approximately twenty-five.

The command files in MBDS version two are UNIX *makefiles*. These have a capability for tracking which source-code files have been modified (and which have not) since the last time a program's source-code files are compiled and linked. Makefiles also allow the programmer to state which source-code files are *dependent* on other source-code files. Through makefiles, only modified source-code files are recompiled, and they (and files dependent on them) are relinked. Some of the MBDS processes require over thirty-five source-code files, and each shares some common source-code files with the other MBDS processes. Therefore,

the management of the MBDS software is easier, and modifications can be accomplished faster, when makefiles are used.

Version three of MBDS uses the VMS operating system which does not have a makefile utility available. The VMS operating system provides command files, but they do not have the capability of tracking source-code modifications and dependencies. Therefore, to use command files for managing the implementation, not only must all of the makefiles be rewritten but the logic of their organization must also be changed. This means that MBDS version three requires over fifty command files to manage its implementation, more than twice the number of makefiles used by version two. Program modifications also take longer since it is easier to recompile and link all source-code files needed for a program (even unmodified ones), than to manually keep track of which have changed and what files are dependent on them.

In order to avoid the problems caused by not using makefiles, an initial attempt has been made to utilize the EUNICE environment (on the VAX-11/780). In EUNICE (a software system that emulates UNIX) makefiles can be used to compile programs which run on the VMS operating system. However, while the programs run on the VAX-11/780 (with VMS) they prove to be not downward-compatible to the Micro-Vaxes, which are used for the MBDS backends. (The Micro-Vaxs are designed to be *upward*-compatible to the VAX-11/780, but not vice versa.) Therefore, the makefiles had to be redone as command files.

The MBDS command-files are converted one set at a time, as each set is converted an attempt is made to compile the MBDS process. While it is known (because of the MBDS design) which files need changes, there is a desire to see if any unexpected compile-time errors might occur. No compile-time errors emerged

some *warning* messages have been generated because of variable names which contain too many characters. These are truncated by the compiler but do not cause any problems (the truncated length happens to be long enough to maintain uniqueness). Once each of the processes is compiled, what essentially exists is a version of MBDS with program segments executable on the VMS operating system, but still only able to communicate with each other using the UNIX communication facilities. Obviously, the next step in the porting is to modify the source code which performs communications.

As we stated in previous chapters, MBDS performs two types of communication, intra-computer and inter-computer. With the UNIX operating system (used by MBDS version two) both types of communications are accomplished in the same manner. The VMS operating system (used by MBDS version three) uses different means to accomplish each of the two types of communication. The decision is made to modify the MBDS communications software in two phases. In the first phase, an intermediate version of MBDS is created which utilized only one computing system, supporting both the controller and one backend. From a functional point of view, the controller and each of the backends does not know where the other components are executing.

Since the intermediate version only used one computer, there is only a need to modify the intra-computer communications at that point. Version three of MBDS uses VMS mailboxes for intra-computer communications in the controller and backends (like the version one controller). For the intermediate version, the sections of code which normally perform inter-computer communications can call the same virtual communication interfaces as the intra-computer communications.

When an attempt is made to test the intermediate MBDS version the first unexpected problems arise. After a considerable debugging effort the problem is

isolated to three lines of source code. The problem is *isolated* in the sense that it is known to occur in a particular line, but it is not obvious why it occurred. By all observations the code is correctly written, and it has functioned properly, as is, in version two. Eventually, it is suggested that the code might be correct to the human observer but is being incorrectly interpreted by the compiler. The lines of code in question use some short-hand coding techniques, which has apparently complicated parsing them. The lines of code causing the problem looked like this (the problem areas are in **bold**):

```
while (msg_q[(*index)] != 0)
  data[j++] = msg_q[(*index)++];

data[j] = msg_q[(*index)++];
```

They are modified, to remove the short-hand techniques, resulting in the code shown below (note the need for two additional lines of code):

```
i = *index;
while (msg_q[i] != 0)
   data[j++] = msg_q[i++];

data[j] = msg_q[i++];
*index = i;
```

Once these changes are made the section of code functions properly, similar changes are needed in three other sections of the source-code. These are the only changes that have been required to get the intermediate version of MBDS working properly.

The next step is to modify the inter-computer computer communications. Since the UNIX version does not have separate interfaces for inter-computer communications, these are not actually *modifications*. Instead, entirely new interfaces

must be created for VMS inter-computer communications. MBDS version three uses DECNET software to perform the inter-computer communication. These interfaces (as well as the ones for the VMS intra-computer communication) are discussed further in the next section. After the inter-computer communication interfaces are completed the only task remaining is to perform some final testing. No additional modifications are necessary to create MBDS version three.

B. THE MBDS COMMUNICATION INTERFACES

Before proceeding further with any discussion of modifying the MBDS communication interfaces, we first present a general overview of the MBDS software architecture and how it utilizes the communication interfaces. As we have stated before, each MBDS implementation has a controller and one or more backends. The controller is comprised of six processes, and each of the backends has five processes. In a normal implementation the controller (i.e., its six processes) is run on its own computing system, there is also a separate computing system for each backend. Since each process is an independently run program it has no direct connection to any other process, yet the processes must pass messages (information and data) to each other for MBDS to operate. Within each process there are the virtual interfaces for sending and receiving messages, it is these *send* and *receive* routines which must actually be modified whenever MBDS is ported. The send and receive routines only perform the intra-computer communication.

Of the six processes in the controller only four are needed by MBDS for database operations, the other two processes are used for inter-computer communication. The same is true for each backend, that is, only three of the backend processes are used for database operations, the other two are for inter-computer communication. These processes (in the controller and each backend) operate as

the virtual interfaces for inter-computer communication. To avoid confusion with the intra-computer interfaces we call these processes *get-net* and *put-net* (as opposed to send and receive). Excluding the get-net and put-net processes, all MBDS database-processes perform only intra-computer communication. The pseudocode for any MBDS database-process (at a very high level) looks like this :

```
While MBDS is operating do
    Receive a message
    Perform any required processing
    Send any required response
```

From looking at the pseudo code you might wonder how messages ever get from one backend to another, or from a backend to the controller. The answer is in the last line of the pseudo code, the key word is *required.* The required response may mean sending a message to more than one other process, if one of those processes is the put-net process then the message eventually goes to a process on another computing system. Therefore, the pseudocode for the put-net process looks like this:

```
While MBDS is operating do
    Receive a message
    Put it to the appropriate process
```

The appropriate process to which a message is *put* is always a get-net process on another computer system. The pseudo code for a get-net process looks like this:

```
While MBDS is operating do
    Get a message
    send it to the appropriate process
```

Now lets examine how the MBDS intra-computer communication interfaces (send and receive) have been modified to operate under the VMS operating system.

1. Modifying the Intra-Computer Communications

The controller for MBDS version one uses the same VAX-11/780 as the controller for MBDS version three. While version one runs on VMS 3.7 and version three runs on VMS 4.2, the intra-computer communication facilities provided by the two VMS releases are the same. Therefore, the MBDS version one controller communication interfaces can be used for the MBDS version three controller without modifications, and can be used for the version three backends with only minor modification.

In the VMS operating system, intra-computer communication is accomplished with VMS *mailboxes*. These mailboxes are virtual devices which can be created through the operating system. When a program calls the operating system to create a mailbox, the program must specify a logical name to be assigned the mailbox. The operating system then creates a mailbox, with the specified logical name, and provides the program with a logical channel to the mailbox. Once a mailbox is created messages may be put into it (or taken out of it) by writing to (or reading from) the logical channel provided by the system. If a call is made to the operating system to create a mailbox with a logical name that has already been used, no new mailbox is created, the operating system just provides another logical channel to the already existing mailbox. This is how different processes on the same computer can communicate. If several processes all use the same logical name in a create-mailbox call to the operating system, then they all share the same mailbox.

The MBDS intra-computer communication (using mailboxes) is accomplished by having each process in the controller (or backend) create a mailbox with its own logical name, and mailboxes with the logical names for each of the other processes in the controller (or backend). The logical names are standardized

for all controller (and backend) processes. As we have said, if the same logical name is used more than once, multiple mailboxes are not created. Rather, only multiple channels to the mailbox with that logical name are created. Therefore, when all processes in the controller (or backend) have finished with their create-mailbox calls, there is *one* mailbox for each process in the controller (or backend). Each process has a logical channel to its own mailbox, and logical channels to the other processes' mailboxes. The protocol that MBDS uses to make this scheme work is that processes can only read their own mailboxes, and can only write to other processes' mailboxes. There is no need to write to their own, or read from any one elses.

The MBDS version one controller's intra-computer communication software is used without changes for the controller in MBDS version three. To create the version three backend intra-computer communication software, it is only necessary to use a copy of the controller software with the controller logical names (for the mailboxes) changed to backend logical names.

To create the intra-computer communications software for the intermediate MBDS version (i.e., where the controller and backend run on a single computer together), one additional (temporary) change is made. An additional mailbox is created for the get-net and put-net routines in the backend and the controller. (Actually, only additional channels to existing mailboxes are created.) Normally, a controller process only has channels to mailboxes for controller processes, and backends only have channels to backend processes. For the intermediate version, put-net in the controller had a channel to get-net's mailbox in the backend, and vice versa. This allowed the get-net and put-net routines to use the same virtual interfaces that are used by the send and receive routines (which perform all of the intra-computer communication). For the final version of

MBDS (version three). put-net and get-net no longer use mailboxes for communication. This is because the VMS operating system does not allow logical channels to be established to a mailbox in another computer.

2. Creating the New Inter-Computer Communications

MBDS version three, like version two, uses Ethernet communication hardware to connect the controller to the backends (and the backends to each other). However, from a functional viewpoint, MBDS version three's inter-computer communication software operates more like MBDS version one (which used point-to-point communication hardware). This is because the way DECNET (VMS) communication software operates it does not provide a broadcasting capability. DECNET is the operating system software used for the version three inter-computer communications.

In the UNIX (Ethernet) environment all processes that wish to communicate simply associate themselves with the network, each process then has a logical communication link with all of the other processes that are associated with the network. (This is much the same as when processes under VMS associate themselves with a common mailbox, for intra-computer communication.) However, in the DECNET (Ethernet) environment each process must establish a separate logical link for every other process it wants to communicate with, therefore, this is essentially (software) point-to-point communication.

When using the DECNET software, all processes which communicate with each other are either *source* processes, or *target* processes. A source process is one that initiates a request to establish a communication link with another process. the target process is the one that receives the request. A given process can be the target for a communication link with one process, and the source for a communication link with another. The DECNET software requires that a target

process is in execution before its source process requests the communication link. A requirement like this does not exist for inter-computer communication in MBDS version one. or version two (nor does it exist for intra-computer communication. in any version). Therefore, the logic for the put-net and get-net processes in version three must be completely changed.

In order to establish, and maintain. communication under the DECNET software the get-net and the put-net processes must operate under the following constraints:

- any process which is a target must be executing before its source process attempts to establish communication.
- if a process is to have multiple communication links, where it is a target in some cases. it must be capable of maintaining communication on established links. while still waiting for (or accepting) any links where it is the target,
- a target process must inform the operating system (through DECNET) of the *network* name it is using, so that DECNET can properly route connection requests,
- a source request must know the network name for all of its target processes. and it must know the name of the computer (the *nodename*) that each target executes on.
- a process must know the total number of communication links it should have. how many it is : target for. and how many times it is a source. (This is because MBDS configures the number of backends to be used. on-the-fly. when it is started.)

Now let us review how the get-net and the put-net routines for MBDS version three were organized to meet these criteria.

In the MBDS controller, both the get-net and the put-net processes are always source processes. Therefore, when MBDS is initially started. the controller processes are executed after all of the backend processes are executed. This satisfies the target/source ordering for the controller-to-backend communication. As soon as the controller is informed (by the user) how many backends are to be used the get-net and the put-net routines can begin establishing (requesting) communication links. Since the controller (get-net and put-net) communication

routines are never targets, they do not have to inform DECNET of any network names (they do not need network names). Also, since they are never targets, they do not have a requirement to accept a connection request while communicating (they never accept connection requests). When requesting the communication links, get-net and put-net know the names of the processes they communicate with because the MBDS process names are standardized. In order for them to know the computer systems (nodenames) that the backends are executing on, the nodenames are standardized as well.

Each nodename consists of a character string that ends with a number (e.g., CSMV1, or CSMV2), only the numbers at the end are different. The convention used by MBDS is that, if only one backend is used, it is on the computer with the lowest nodename (i.e., CSMV1), if multiple backends are used they are on the computers with the lowest consecutive nodenames (e.g., for three backends: CSMV1, CSMV2, and CSMV3). As soon as the controller routines know the number of backends, they can construct the proper nodenames.

For the MBDS backend-to-backend, inter-computer communication the backend get-net processes are always targets, and the backend put-net routines are always the source. Remember that backend put-net processes are *targets* for controller-to-backend communication, they are the only processes which perform both as targets, and sources, depending on the situation. Since there is always a single controller, the backend put-net process knows it is a target for a single communication link. The backend put-net process does not begin requesting links (acting as source) to the other backends get-net processes until after it has received its first message from the controller. By convention, this first message is always the number of backends to be used. The put-net process then knows how many communication links it must have, and where it is the source. The put-net

process forwards this message (using the intra-computer communication) to its respective get-net process and then the get-net process knows how many times it must act as a target for a communication link. The backend put-net routines determine nodenames (for the other backends computer systems) in the same manner used by the controller. When the backend get-net and put-net routines are first executed, they inform DECNET of the (standardized) network names that are being used.

Because the backend communication processes (get-net and put-net) act as target tasks with multiple communication links, they must be capable of maintaining the communications on any established links while waiting for (or accepting) connection requests. They accomplish this by using VMS mailboxes, these are the same types of mailboxes used for intra-computer communication. However, in this case, the mailboxes are not used for MBDS messages, but are used for DECNET status information. When a process uses DECNET to perform communication it establishes a logical link to DECNET. This is in addition to the logical links for the other processes that it communicates with. Any time a process establishes a logical link, the process has the option of associating a mailbox with that logical link. One mailbox may be associated with multiple links.

If a communication link has a mailbox associated with it, DECNET places network status messages (about that link) into the mailbox. These status messages may include information such as the fact that DECNET has a connection request for a process, the fact that an established link has received a message, or the fact that a process on the other end of a communication link has disconnected itself. Therefore, the backend communication processes can handle multiple links by associating a single mailbox with all of their communication links. Each time the process completes an operation (e.g., putting a message, getting a

message. or accepting a connection request) it then reads its mailbox. The next message in the mailbox determines what operation the process must perform next. DECNET automatically queues multiple messages in the mail box. If a process' mailbox is empty the process waits until a message arrives, that is, an empty mailbox means there is nothing else for a process to do.

Mailboxes are used for managing multiple connections by all of the MBDS inter-computer communication processes which act as targets, or as message-receiving processes. Therefore, the put-net process in the controller is the only one that does not need a mailbox to manage its communications (it is never a target, and only puts messages). By convention, any time a process puts an inter-computer MBDS message to another process it also puts a DECNET notification message to the same process (these are optional with DECNET, and not generated automatically). The MBDS message goes on the logical communication link, and the notification message is placed (by DECNET) into the mailbox for that communication link, at the receiving process. If the DECNET notification message were not sent the receiving process would never be prompted to look at the communication link for a message.

V. SUMMARY AND CONCLUSIONS

In this thesis we have focused on three major areas of work. First, a paradigm for a software system has been presented, and we have examined how the components in the paradigm relate to the portability of the software system. Let us review the general issues of software portability discussed in the previous chapters. We have presented a paradigm for a software system which shows that there are two main components to any software system, the system source code and the operatings system commands used by the software system. Each of these main components has additional sublevels. For the system source code there are the sublevels of machine code, assembler language and high-level code. The sublevels of operating system commands are, basic commands, command files, and utilities. We have also stated that there are three parts to a computing environment which a software system interacts with, the hardware, the translator (used by the source code for the software system) and the operating system. Finally, we have stated that, the components of a software system interact with the parts of its computing environment through three types of processing, basic processing, translator environment processing, and runtime environment processing.

Our analysis showed that the ease with which a software system reacts to porting is determined by three factors:

- the levels of system source code and operating system commands used to construct the system,
- the types of processing which the software system performs, and
- the parts of the computing environment which are changed.

In general, a software system is highly portable if it uses only high-level code, and if it avoids runtime environment processing whenever possible.

The second focus of the thesis has examined how well the MBDS software stands up to porting. To construct its three versions, the MBDS software has been subjected to all three types of porting. Each time the system was ported, MBDS ran on new hardware, had a new operating system and had a new translator for its system source code. Despite the drastic changes in its environment, MBDS performed extremely well each time it was ported. In neither case did the *fundamental* MBDS design require modification. This can be attributed to several factors. First, the MBDS software is written entirely in high-level code and the use of runtime processing has been avoided except where absolutely necessary (e.g., communications and disk input/output). Additionally, any processing which might complicate portability has been buffered in MBDS through the use of abstraction, encapsulation and virtual interfaces.

The third focus of the thesis has involved the details of the MBDS porting. The major work of porting MBDS has involved modifying the sections of code which create the virtual interfaces. The *interfaces* themselves have not been changed, only the way in which the interfaces accomplished their jobs are modified. These changes have involved the MBDS communication software. For the inter-computer communications, the MBDS version one software has been modified to create MBDS version three. This intra-computer communication is accomplished using VMS mailboxes. For the inter-computer communications, entirely new communication software has been created in MBDS version three. The VMS DECNET communication drivers are utilized for the inter computer communications. (Source code for the new inter-computer communication software is contained in the appendixes to the thesis.)

In addition to modifying the communication interfaces, creating MBDS version three has also required modifying all of the operating system command files used to manage the software. This involved rewriting the UNIX makefiles (from MBDS version two) as VMS command procedures (for MBDS version three). Rewriting the command files has been the most time consuming task in the porting process.

Overall, MBDS has proven to be easily portable. With the exception of some minor problems caused by the way the new compiler parsed some lines of code (in MBDS version three), all of the modifications required to port MBDS have been known before the porting began. Additional information on the portability of MBDS can be found in [Ref. 1] which deals with the process of porting MBDS version one to MBDS version two.

```
/*****************************************/
/*                                       */
/*     V A X / V M S   G _ P C L C       */
/*                                       */
/*****************************************/


#include  <stdio.h>      /* Standard I/O definition */
#include  <ssdef.h>      /* VMS return status codes */
#include  <iodef.h>      /* VMS I/O return codes    */
#include  <msgdef.h>     /* DECNET msg definitions  */
#include  <dvidef.h>     /* VMS device definitions  */
#include  "commdata.def" /* MBDS common defs        */
#include  "msg.def"      /* MBDS msg-type defs      */
#include  "flags.def"    /* MBDS compile flags      */
#define   MAXBE  16      /* max num of backends     */


char     netbuf[MSGLEN];      /* network buffer      */
char     mbxbuf[MSGLEN];      /* mailbox buffer      */
char     msg[MSGLEN];         /* MBDS-msg buffer     */

short    net_chan;            /* channel to DECNET   */
short    mbx_chan;            /* channel to mailbox  */
short    be_chan[MAXBE + 1];  /* chan's to backends  */
                              /* backend 0 is not used */

short    be_dev[MAXBE + 1];   /* device num of chans */
short    NoBackends;          /* the num of backends */

struct     msg_hdr    head;
```

```
main ()
  {
  int  i.                    /* loop index           */
       StopSys:              /* system running flag */

  char NoBEs [NoBElength + 1]:

  /* init intra-computer communication */
  mbinit( G_PCLC ):

#ifdef EnExFlag
        printf("Enter GPCL\n");
#endif

   /* get the number of backends */

#ifdef pr_flag
       printf("Getting number of backends\n"):
#endif
   receive(msg. &head):

#ifdef pr_flag
       m_prnt(msg. &head);
#endif

   if (head.type != SetNoBEs)
      {
      printf("** Error in GPCLC. first msg must be "):
      printf("of type SetNoBEs **\n");
      printf("** Type = %d **\n". head.type);
      m_prnt(&msg. &head);
      sleep(DELAY):
      exit():
      }

   /* Extract the number of backends */
   for (i=0; ((NoBEs[i] = msg[i]) != '\0'): i++):

   NoBackends = str_to_num(NoBEs):

#ifdef pr_flag
       printf("Number of backends= %d\n". NoBackends):
#endif
```

```
    /* Init network connections to P_PCL's in backends */
    net_init(NoBackends);

          StopSys = FALSE;
          while ( !StopSys )
            {
             get_message(msg,&head);


             switch (head.type)
                {

/* msg type for msgs from DMgt (BEnds) to IIG (ctller) */
                  case(ReqForNewDescId):
                  head.receiver = IIG;
                  break;

                 case(ClusId):
                  head.receiver = IIG;
                  break;

/*  msg type for msgs from RecP (BEnds) to PP (cntrl)  */
                 case(BC_Res):
                  head.receiver = PP;
                  break;

                 case(BC_AO_Res):
                  head.receiver = PP;
                  break;

/* msg type for msgs from RecP (BEnds) to ReqP (cntrl) */
                 case(RetFetCausedByUpdRes):
                  head.receiver = REQP;
                  break;

                 case(RecChangedClus):
                  head.receiver = REQP;
                  break;

                 case( NoMoreGenIns ):
                  head.receiver = REQP;
                  break;
```

```
/* msg type for msgs from BEnds to TI (ctller) */

                case(Error):
                 head.receiver = TI:
                 break:

                case(ErrorFree):
                 head.receiver = TI;
                 break;

                 case(GeTimes):
                  head.receiver = TI;
                  break;

                 case(Tim_Arr_Emp):
                  head.receiver = TI:
                  break;

                 case(Stop):
                  StopSys = TRUE:
                  break;

                default:
                 printf("Invalid message type ");
                 printf("encountered: %d n",head.type):
                 exit_gracefully():

               }/* end switch */

            if ( !StopSys )
               {
                 head.sender = G_PCLC;
                 send(msg, &head);
#ifdef pr_flag
            m_prnt (msg, &head):
#endif
               }

           }/* end while */

#ifdef EnExFlag
        printf("Exit GPCL\n");
#endif
       }
```

```
/* routine to get the next message from the network */

get_message(mptr,hptr)

char    *mptr;
struct msg_hdr *hptr;
  {
int     stat,
        msglen,
        func,
        j, k;

short iosb[4];
char  rcvbuf[MSGLEN + 1],
        intbuf[2] = "\0",
        tmpstr[5];

short msg_rec;

#ifdef EnExFlag
        printf("Enter get_message\n");
#endif

     /* Check mailbox for message notices */
     read_mbx(mbxbuf);

     /* Read the message */
     read_net(rcvbuf, mbxbuf);

     /*  Get the header information  */

     k=0;

    /* get sender */
    for (j=0; j < 3; j++)
         tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->sender = str_to_num(tmpstr);

    /* get receiver */
    for (j=0; j < 3; j++)
         tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->receiver = str_to_num(tmpstr);
```

```
    /* get the type */
    for (j=0; j < 3; j++)
        tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->type = str_to_num(tmpstr);

#ifdef pr_flag
        m_prnt(rcvbuf, hptr);
#endif

    /*  copy the message  */
    j=0;

    while ( (k < MSGLEN) && (rcvbuf[k] != EOMsg) )
        mptr[j++] = rcvbuf[k++];

    mptr[j] = rcvbuf[k];      /* get EOMsg */

    if (k >= MSGLEN)
       printf("***** Value of MSGLEN ");
       printf("should be increased *****\n");

#ifdef EnExFlag
        printf("Exit get_message\n");
#endif

  }   /* end get_message */
```

```
/* Routine to check the mailbox for message notices */

read_mbx(buf)

char *buf;
{
short  msg_rec;
short  stat;
short  iosb[4];

#ifdef EnExFlag
       printf("Enter read_mbx\n");
#endif

   msg_rec = FALSE;

   while ( ! msg_rec )
     {
#ifdef pr_flag
       printf("Calling qio, read mailbox\n");
#endif
     stat= sys$qiow(0, mbx_chan, IO$_READVBLK,
                    iosb, 0,0, buf, MSGLEN, 0,0,0,0);
#ifdef pr_flag
       printf("Returned from qio \n");
#endif
     if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
       {
       printf("** Mailbox read error, stat= %d (%x), ");
       printf("iosb[0]= %d (%x) **\n", stat, stat,
                                       iosb[0], iosb[0]);
       exit_gracefully();
       }

     switch (buf[0])
       {
       case MSG$_INTMSG:
                          {
                          msg_rec = TRUE;
                          break;
                          }
```

```
        case MSG$_CONNECT:
                        {
                        printf("** Network connection ");
                        printf("requested **\n");
                        printf("** GPCLC should not ");
                        print("receive connect req's **\n");
                        exit_gracefully();
                        }

        case MSG$_CONFIRM: break;

        default:        {
                        printf("** Network error, ");
                        printf("mbxbuf[0]= %d (%x) **\n",
                                        buf[0], buf[0]);
                        exit_gracefully();
                        }
        }  /* end switch */
      } /* end while */

#ifdef EnExFlag
        printf("Exit read_mbx\n");
#endif

}  /* end read_mbx */
```

```
/*   Routine to read a message from a backend   */

read_net (rcvbuf, mbxbuf)

char *rcvbuf, *mbxbuf;
  {
  short *unit,
        BEnum,
        func,
        stat,
        iosb[4];

#ifdef EnExFlag
        printf("Enter read_net\n");
#endif

  unit= mbxbuf;

  for (BEnum = 1; be_dev[BEnum] != unit[1]; BEnum++)
    {
    if (BEnum == NoBackends)
        {
        printf("*** Cannot locate unit number to ");
        print("read net with. ***\n");
        printf("*** Searched through %d backends.");
        printf("                  ***\n", BEnum);
        exit_gracefully();
        }
    }

#ifdef pr_flag
        printf("Calling qio read from backend %d\n",BEnum);
#endif

    func = IO$_READVBLK;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb,
                                0,0, rcvbuf, MSGLEN, 0,0,0,0);

#ifdef pr_flag
        printf("Returned from qio\n");
#endif
```

```
    if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
      {
      printf("** Read error be_chan %d. stat= %d(%x), ");
      printf("iosb[0]= %d(%x) **\n", BEnum, stat,
                                     stat, iosb[0], iosb[0]);
      exit_gracefully();
      }

#ifdef EnExFlag
       printf("Exit read_net\n");
#endif

  }  /* end read_net  */
```

```
/* Routine to initializ Decnet links to each backend */

net_init(NoBEs)

int NoBEs;
{
  int   stat, i;
  short iosb[4];
  char  nodespec[128], tmpstr[5];

  struct sd {
            int  len;
            char *ptr;
            }
            netnam = { 5, "_NET:"  },
            ncb    = { 0, nodespec },
            netmbx = { 6, "NETMBX" };

  long      dviunit[1];
  short     dviunit_len[1];

  /* structure to get unit numbers for chan's to net  */
  struct    {
            short  len;       /* buffer length          */
            short  code;      /* item code              */
            long   *unit;     /* addr to return unit to */
            short  *unitlen;  /* length of unit         */
            long   nul;       /* end of descriptor      */
            }
            dvi = {4, DVI$_UNIT, dviunit, dviunit_len,0};

#ifdef EnExFlag
       printf("Enter net_init\n");
#endif

  /* create a mailbox */
  stat=sys$crembx(0, &mbx_chan, MSGLEN, MSGMAX,
                                        0, 0, &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Error creating mailbox, ");
     printf("stat= %d (%x) **\n", stat, stat);
     exit_gracefully();
     }
```

```
  /* assign channels to the net */
  for(i=1; i <= NoBEs; i++)
    {
    stat= sys$assign(&netnam, &be_chan[i], 0, &netmbx);
    if (stat != SS$_NORMAL)
       {
       printf("** Error in assign be_chan %d, ");
       printf("stat= %d (%x) **\n", 1,stat.stat);
       exit_gracefully();
       }
     } /* end for i */

  /* Establish logocal link */
  for(i=1; i <= NoBEs; i++)
    {
    /* build network connect block */
    strcpy(nodespec, "CSMV");
    num_to_str(i, tmpstr);
    strcat (nodespec, tmpstr);
    strcat (nodespec, "::\"0=PPCLB\"");

#ifdef pr_flag
  printf("backend %d nodespec, \"%s\"\n", i, nodespec);
#endif

    ncb.len= strlen(nodespec);

    /*  Request the connection */
    stat= sys$qiow(0, be_chan[i], IO$_ACCESS, iosb,
                                    0,0,0, &ncb, 0,0,0,0);
    if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
       {
       printf("** Access error be_chan %d, stat= %d ");
       printf("(%x), iosb[0]= %d (%x) **\n", i, stat,
                                stat, iosb[0], iosb[0]);
       exit_gracefully();
       }
     } /* end for i */
```

```
  /* Get unit numbers for the channels */
  for(i=1; i <= NoBEs; i++)
    {
    stat= sys$getdvi(1, be_chan[i], 0, &dvi.iosb,0,0,0);
    if (stat != SS$_NORMAL)
        {
        printf("** Error getting channel number for BE ");
        printf("%d, stat= %d (%x) **\n", i, stat, stat);
        exit_gracefully();
        }
    sys$waitfr(1);
    if (iosb[0] != SS$_NORMAL)
        {
        printf("** Error getting channel number for ");
        printf("BE %d, iosb[0]= %d(%x) **\n", i,
                                        iosb[0], iosb[0]);
        exit_gracefully();
        }
    be_dev[i] = *dviunit;

    } /* end for i */


#ifdef EnExFlag
        printf("Exit net_init\n");
#endif

}  /*  End net_init  */
```

```
/* routine to disconnect all network links */

disconnect()
{
 int stat, i;

#ifdef EnExFlag
        printf("Enter Disconnect\n");
#endif

 for (i=1; i <= NoBackends; i++)
    {
#ifdef pr_flag
        printf("Disconnecting backend %d\n", i);
#endif
    stat= sys$dassgn(be_chan[i]);
    if (stat != SS$_NORMAL)
     printf("** Dassign Error for backend %d, ");
     printf("stat= %d (%x) **\n",i,stat, stat);
    }
#ifdef EnExFlag
        printf("Exit Disconnect\n");
#endif

}  /* End disconnect */



/* Routine to close network connections, then abort */

exit_gracefully()
{
    sleep(DELAY);
    disconnect();
    exit();
}
```

```
/*****************************************/
/*                                       */
/*     V A X / V M S    P _ P C L C      */
/*                                       */
/*****************************************/


#include  <stdio.h>      /* Standard I/O definition */
#include  <ssdef.h>      /* VMS return status codes */
#include  <iodef.h>      /* VMS I/O return codes    */
#include  <msgdef.h>     /* DECNET msg definitions  */
#include  "commdata.def"  /* MBDS common defs       */
#include  "msg.def"       /* MBDS msg-type defs     */
#include  "flags.def"     /* MBDS compile flags     */
#include  "beno.dcl"      /* backend num decl       */

#define   MAXBE   16      /* max num of backends    */
#define   TRUE    1
#define   FALSE   0

char      netbuf[MSGLEN];     /* network buffer     */
char      msg[MSGLEN]:        /* MBDS-msg buffer    */

short     be_chan[MAXBE + 1]; /* chan's to backends */
                              /* backend 0 is not used */

short     be_dev[MAXBE + 1]; /* device num of chans */
short     NoBackends;        /* the num of backends */

struct    msg_hdr    head;
```

```
main ()
    {
     int StopSys,i,j,k;
     char NoBEs[NoBElength + 1 ];

#ifdef EnExFlag
     printf( "Enter PPCL\n" );
#endif

     /* init intra-computer communication       */
     mbinit( P_PCLC );

     /* receive a message from a controller     */
     /* task into ppcl's mailbox                */
     receive(&msg[0], &head);

     /* The first message should type SetBEno. */
     if( head.type != SetNoBEs )
       {
         printf( "Error in PPCL. 1st message ");
         printf("must be of type SetNoBEs" );
         printf( "  Type = %d\n", head.type );
         msend(&msg[0], &head);
         abort();
       }

     /*  send num of BE to GPCLC */
     head.sender   = P_PCLC;
     head.receiver = G_PCLC;
     send(&msg[0], &head);

     /* Extract the NoBackends */
     for( k=0, j=0; (NoBEs[j] = msg[k]) != '\0';
                                    k++, j++ );
         NoBackends = str_to_num( NoBEs );

         k++;

     /* Initialize connections to backends  */
     net_init(NoBackends);

     /* send BACKEND_NO and NoBackends to backends */
     /* Change the message type */
           head.type = SetBEno;
```

```
    /* send the msg to each BE */
    for (i=1; i <= NoBackends; i = i + 1)
        {
         /* Put the backend number into message */
         len_num_to_str( (i), NoBElength, &msg[k] );
         msg[k + NoBElength +1] = EOMsg;

         /* send the msg to the specified BE */
         head.receiver= G_PCLB;
         put_message(msg,&head, i);

        }

 /* receive message from a controller */
 /* task into ppcl's mailbox          */
 receive (&msg[0], &head);

 StopSys = FALSE;
 while ( !StopSys )
   {

    /* send the msg to each BE */
    for (i=1; i <= NoBackends; i = i + 1)
        {
         /* send the msg to the specified BE */
         put_message (msg,&head,i);
        }

    if ( head.type == Stop )
          StopSys = TRUE;
    else
         /* receive the next message */
         get_message( &msg[0], &head);

   }/* end while */

 exit_gracefully();
}
```

```
/* routine to send message over network to a backend */

put_message(mptr,hptr,BEnum)

   char    *mptr;
   struct msg_hdr *hptr;
   int     BEnum;

   {
    int    stat,
           msglen,
           func,
           j, k;

    short iosb[4];
    char  sndbuf[MSGLEN + 1],
          intbuf[2] = "\0", /* null message */
          tmpstr[5];


#ifdef EnExFlag
        printf("Enter put_message\n");
#endif

     hptr -> sender =   P_PCLC;
     hptr -> receiver = G_PCLB;

     k=0;

     /* copy header into message to be sent */

     len_num_to_str(hptr->sender, 3, tmpstr);
     for (j=0; j < 3; j++)
         sndbuf[k++] = tmpstr[j];

     len_num_to_str(hptr->receiver, 3, tmpstr);
     for (j=0; j < 3; j++)
         sndbuf[k++] = tmpstr[j];

     len_num_to_str(hptr->type, 3, tmpstr);
     for (j=0; j < 3; j++)
         sndbuf[k++] = tmpstr[j];
```

```
    /*  copy the message  */
    j=0;
    while ( (k < MSGLEN) &&
          ((sndbuf[k++] = mptr[j++]) != EOMsg) );


    if (k >= MSGLEN)
       printf("***** Value of MSGLEN ");
       printf("should be increased *****\n");

    msglen = k;

    /* send the message */

#ifdef pr_flag
    printf("Calling qio write to backend %d\n", BEnum);
    m_prnt(sndbuf, hptr);
#endif

    func = IO$_WRITEVBLK;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb, 0,0,
                                  sndbuf, msglen, 0,0,0,0);
#ifdef pr_flag
       printf("Returned from qio, write\n");
#endif

    if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
      {
      printf("** Write error be_chan %d, stat= %d ");
      printf("(%x), iosb[0]= %d (%x) **\n", BEnum,
                           stat, stat, iosb[0], iosb[0]);
      exit_gracefully();
      }


   /* let receiver know msg is there */
#ifdef pr_flag
       printf("Calling qio, interrupt\n");
#endif

    func = IO$_WRITEVBLK | IO$M_INTERRUPT;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb, 0,0,
                                     intbuf, 1, 0,0,0,0);
```

```
#ifdef pr_flag
        printf("Returned from qio. interrupt \n");
#endif

     if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
       {
      printf("** Interrupt error be_chan %d. ");
      printf("stat= %d(%x), iosb[0]= %d(%x) **\n",
                      BEnum, stat. stat. iosb[0], iosb[0]);
       exit_gracefully();
       }

#ifdef EnExFlag
        printf("Exit put_message\n");
#endif

   }
```

```
/* Initializes Decnet links to each of the backends */

net_init(NoBEs)

int NoBEs;

{
  int   stat, i;
  short iosb[4];
  char  nodespec[128], tmpstr[5];

  struct sd {
            int  len;
            char *ptr;
            }
            netnam = { 5, "_NET:" },
            ncb    = { 0, nodespec };

#ifdef EnExFlag
        printf("Enter net_init\n");
#endif

  for(i=1; i <= NoBEs; i++)
    {
    /* assign a channel to the net */
    stat= sys$assign(&netnam, &be_chan[i], 0, 0);
    if (stat != SS$_NORMAL)
       {
       printf("** Error in assign be_chan %d, ");
       printf("stat= %d (%x) **\n", l.stat.stat);
       exit_gracefully();
       }
     }

  /* Establish logocal link */
  for(i=1; i <= NoBEs; i++)
    {
    /* build network connect block */
    strcpy(nodespec, "CSMV");
    num_to_str(i, tmpstr);
    strcat (nodespec, tmpstr);
    strcat (nodespec, "::\"0=GPCLB\"");
```

```
#ifdef pr_flag
  printf("backend %d nodespec.  "%s " n", i, nodespec);
#endif

    ncb.len= strlen(nodespec);

    /*  Request the connection */
    stat= sys$qiow(0, be_chan[i], IO$_ACCESS, iosb,
                                        0,0,0, &ncb, 0,0,0,0);
    if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
       {
       printf("** Access error be_chan %d, stat= %d ");
       printf("(%x), iosb[0]= %d (%x) **\n", i, stat,
                                    stat, iosb[0], iosb[0]);
       exit_gracefully();
       }

     } /* end for i */

#ifdef EnExFlag
       printf("Exit net_init\n");
#endif
}
```

```
/* routine to disconnect all network links */

disconnect()
{

 int stat, i;

#ifdef EnExFlag
        printf("Enter Disconnect\n");
#endif

 for (i=1; i <= NoBackends; i++)
    {
#ifdef pr_flag
        printf("Disconnecting backend %d\n", i);
#endif
    stat= sys$dassgn(be_chan[i]);
    if (stat != SS$_NORMAL)
     printf("** Dassign Error for backend %d, ");
     printf("stat= %d (%x) **\n",i,stat, stat);
    }
#ifdef EnExFlag
        printf("Exit Disconnect\n");
#endif

}  /* end disconnect */



/* Routine to close network connections, then abort */

exit_gracefully()
{
    sleep(DELAY);
    disconnect();
    exit();
}
```

```
/*****************************************/
/*                                       */
/*     V A X / V M S   G _ P C L B       */
/*                                       */
/*****************************************/

#include  <stdio.h>      /* Standard I/O definition */
#include  <ssdef.h>      /* VMS return status codes */
#include  <iodef.h>      /* VMS I/O return codes    */
#include  <msgdef.h>     /* DECNET msg definitions  */
#include  <dvidef.h>     /* VMS device definitions  */
#include  "commdata.def" /* MBDS common defs        */
#include  "msg.def"      /* MBDS msg-type defs      */
#include  "flags.def"    /* MBDS compile flags      */
#define   MAXBE  16      /* max num of backends     */

char    netbuf[MSGLEN];      /* network buffer      */
char    mbxbuf[MSGLEN];      /* mailbox buffer      */
char    msg[MSGLEN];         /* MBDS-msg buffer     */

short   net_chan;            /* channel to DECNET   */
short   mbx_chan;            /* channel to mailbox  */
short   be_chan[MAXBE + 1]; /* chan's to backends   */
                      /* backend 0 is controller    */

short   be_dev[MAXBE + 1]; /* device num of chans    */
short   NoBackends;        /* the num of backends    */
short   next_chan = 0;     /* the next avail chan    */

struct sd {
          int  len;
          char *ptr;
          }  /* string descriptors for: */

          /* the network device name     */
          netnam = { 5, "_NET:"  },

          /* the mailbox name            */
          netmbx = { 8, "G_NETMBX" };

struct    msg_hdr   head;
```

```
main ()
      {
      int  StopSys:        /* system running flag   */

#ifdef EnExFlag
    printf("Enter G_PCLB\n");
#endif
         /* init intra-computer communication      */
         initsr ( G_PCLB );

         /* set up to use DECNET                   */
         net_init();

         StopSys = FALSE;
         while ( !StopSys )
           {
             /* get a message from the network     */

             get_message(&msg[0], &head):

             /* send the message */
             set_header();
             if( head.type == Stop )
                {              /* exit from MDBS    */
                StopSys = TRUE;

                head.receiver = RECP;
                send(msg, &head);

                head.receiver = DM;
                send(msg. &head):

                head.receiver = CC;
                send(msg. &head):

                head.receiver = P_PCLB:
                send(msg. &head):
                } /* end if part */
```

```
else if( head.type == SetBEno )
    {
    /* set number of backends.       */
    /* and this backend number.      */
    /* GPCL. itself,  does NOT care */
    head.receiver = RECP;
    send(msg. &head);

    head.receiver = DM:
    send(msg, &head);

    head.receiver = CC;
    send(msg. &head):

    head.receiver = P_PCLB:
    send(msg. &head);

  }
else if( head.type ==
    NewDB          ||   /* create new database */
         head.type ==
    Template  ||        /* create new template */
         head.type ==
    SelectDatabase)
    /*assign database to user   */
  {
    /* send to ALL tasks          */
    head.receiver = RECP;
    send(msg. &head);

    head.receiver = DM:
    send(msg. &head);
  }
```

```
#ifdef TimeFlag
                    else if ((head.type >= MIN_RP_MSGTYPE)
                    && (head.type <= MAX_RP_MSGTYPE))
                    {
                        head.receiver = RECP;
                        send(msg,&head);
                    }
                    else if ((head.type >= MIN_CC_MSGTYPE)
                    && (head.type <= MAX_CC_MSGTYPE))
                    {
                        head.receiver = CC;
                        send(msg,&head);
                    }
                    else if ((head.type >= MIN_DM_MSGTYPE)
                    && (head.type <= MAX_DM_MSGTYPE))
                    {
                        head.receiver = DM;
                        send(msg,&head);
                    }

                    else if (head.type == GeTimes)
                          {
                            head.receiver = RECP;
                            send(msg,&head);
                            head.receiver = CC;
                            send(msg,&head);
                            head.receiver = DM;
                            send(msg,&head);
                          } /* end if ( GeTimes ) */
#endif
                    else
                    /* ( != FINISHED &&    != GETIMES )   */
                        send(msg, &head);

            } /* end while */

#ifdef EnExFlag
    printf("Exit G_PCLB\n" );
#endif

          exit();

        } /* end main */
```

```
/* routine to get the next message from the network */

get_message(mptr,hptr)

char    *mptr;
struct msg_hdr *hptr;
  {
int     stat,
        msglen,
        func,
        j, k;

short iosb[4];
char  rcvbuf[MSGLEN + 1],
        intbuf[2] = "\0",
        tmpstr[5];

short msg_rec;

#ifdef EnExFlag
        printf("Enter get_message\n");
#endif

     /*  Check mailbox for message notices */
     read_mbx(mbxbuf);

     /* Read the message */
     read_net(rcvbuf, mbxbuf);

     /*  Get the header information  */

     k=0;

    /* get sender */
    for (j=0; j < 3; j++)
        tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->sender = str_to_num(tmpstr);

    /* get receiver */
    for (j=0; j < 3; j++)
        tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->receiver = str_to_num(tmpstr);
```

```
    /* get the type */
    for (j=0; j < 3; j++)
        tmpstr[j] = rcvbuf[k++];
    tmpstr[j] = '\0';
    hptr->type = str_to_num(tmpstr);

#ifdef pr_flag
        m_prnt(rcvbuf, hptr);
#endif

    /*  copy the message  */
    j=0;

    while ( (k < MSGLEN) && (rcvbuf[k] != EOMsg) )
        mptr[j++] = rcvbuf[k++];

    mptr[j] = rcvbuf[k];      /* get EOMsg */

    if (k >= MSGLEN)
       printf("***** Value of MSGLEN ");
       printf("should be increased *****\n");

#ifdef EnExFlag
        printf("Exit get_message\n");
#endif

  }   /* end get_message */
```

```
/*  Routine to check the mailbox for message notices */

read_mbx(buf)

char *buf;
{
short  msg_rec;
short  stat;
short  iosb[4];

#ifdef EnExFlag
       printf("Enter read_mbx\n");
#endif

   msg_rec = FALSE;

   while ( ! msg_rec )
     {
#ifdef pr_flag
       printf("Calling qio, read mailbox\n");
#endif
     stat= sys$qiow(0, mbx_chan, IO$_READVBLK, iosb,
                          0,0, buf, MSGLEN, 0,0,0,0);
#ifdef pr_flag
       printf("Returned from qio \n");
#endif
     if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
       {
       printf("** Mailbox read error, stat= %d (%x), ");
       printf("iosb[0]= %d (%x) **\n", stat, stat,
                                    iosb[0], iosb[0]);
       exit_gracefully();
       }

     switch (buf[0])
       {
       case MSG$_INTMSG:
                          {
                          msg_rec = TRUE;
                          break;
                          }
```

```
        case MSG$_CONNECT
                        {
                        connect(buf);
                        break;
                        }

        case MSG$_CONFIRM: break;

        default:        {
                        printf("** Network error, ")
                        printf("mbxbuf[0]= %d (%x) **\n",
                                        buf[0], buf[0]);
                        exit_gracefully();
                        }
        }  /* end switch */
      } /* end while */

#ifdef EnExFlag
        printf("Exit read_mbx\n");
#endif

}  /* end read_mbx */
```

```
/*  Routine to read a message from a backend  */

read_net (rcvbuf, mbxbuf)

char *rcvbuf, *mbxbuf;
  {
  short *unit,
        BEnum,
        func,
        stat,
        iosb[4];

#ifdef EnExFlag
        printf("Enter read_net\n");
#endif

  unit= mbxbuf;

  for (BEnum = 0; be_dev[BEnum] != unit[1]; BEnum++)
    {
    if (BEnum >= (next_chan - 1) )
        {
        printf("*** Cannot locate unit number to ");
        printf("read net with.    ***\n");
        printf("*** Searched through %d backends,")
        printf("  next_chan= %d   ***\n", BEnum, next_chan);
        exit_gracefully();
        }
    }

#ifdef pr_flag
        printf("Calling qio read from backend %d\n", BEnum);
#endif

    func = IO$_READVBLK;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb, 0,0,
                                  rcvbuf, MSGLEN, 0,0,0,0);

#ifdef pr_flag
        printf("Returned from qio\n");
#endif
```

```
    if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
      {
      printf("** Read error be_chan %d. stat= %d(%x). ");
      printf("iosb[0]= %d(%x) **\n", BEnum, stat,
                                     stat, iosb[0], iosb[0]);
       exit_gracefully();
       }

#ifdef EnExFlag
       printf("Exit read_net\n");
#endif

  }  /* end read_net  */
```

```
/*  Initialize Decnet to receive connection requests */

net_init()
{
int    stat, i;
short iosb[4];

#define    NFB$C_DECLNAME 0x15
char       nfb[5] = { NFB$C_DECLNAME, 0,0,0,0 };

struct sd {
          int  len;
          char *ptr;
          }
          objnam = { 5, "GPCLB"  },
          nfb_d  = { 5, nfb      };

char       tmpstr    [5];

#ifdef EnExFlag
       printf("Enter net_init\n");
#endif

  /* create a mailbox */
  stat=sys$crembx(0, &mbx_chan, MSGLEN, MSGMAX, 0, 0,
                                                  &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Error creating mailbox, ");
     printf("stat= %d (%x) **\n", stat, stat);
     exit_gracefully();
     }

  /* assign channel to the net */
  stat= sys$assign(&netnam, &net_chan, 0, &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Error in assign for netchan, ");
     printf("stat=%d (%x) **\n", stat, stat);
     exit_gracefully();
     }
```

```
    /* declare a network name */
    stat= sys$qiow(0, net_chan, IO$_ACPCONTROL, iosb,
                        0,0, &nfb_d, &objnam, 0,0,0,0);
    if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
        {
        printf("** Error declaring network name  **\n");
        exit_gracefully();
        }


#ifdef EnExFlag
        printf("Exit net_init\n");
#endif
}
```

```
/* routine to accept a network connection request */

connect(buf)

char *buf;
{
short       .
            offset,
            stat,
            iosb[4];

char        nodespec[128];

struct sd {
            int  len;
            char *ptr;
            }
            ncb     = { 0, nodespec };

long        dviunit[1];
short       dviunit_len[1];

/* structure to get unit numbers for channels to the net */
struct    {
            short  len;       /* buffer length */
            short  code;      /* item code     */
            long   *unit;     /* addr to return unit to */
            short  *unitlen;  /* length of unit */
            long   nul;       /* end of descriptor */
            }
            dvi = { 4, DVI$_UNIT, dviunit, dviunit_len, 0 };

#ifdef EnExFlag
        printf("Enter connect\n");
#endif
  /* see if there are any channels available */
  if (next_chan > MAXBE)
      {
      printf("** Too many connection requests ");
      printf("attempted, next_chan= %d **\n", next_chan);
      exit_gracefully();
      }
```

```
  /* Extract network connect block from mailbox buffer */

  offset  = buf[4] + 5;  /* point to ncb length in buf */
  ncb.len = buf[offset]; /* put length into our ncb    */
  offset++;              /* point past ncb length      */
  for (i=0; i < ncb.len; i++)          /* get the ncb   */
      nodespec[i] = buf [i + offset];
  nodespec[i]= '\0';

#ifdef pr_flag
       printf("** nodespec= %s **\n", nodespec);
       printf("** next_chan= %d **\n", next_chan);
#endif

  /* Assign the next channel to the net */
  stat= sys$assign(&netnam, &be_chan[next_chan], 0,
                                                 &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Assign error be_chan %d, ");
     printf("stat= %d (%x)  **\n",next_chan,stat, stat);
     exit_gracefully();
     }

  /* accept the connection */
  stat= sys$qiow(0, be_chan[next_chan], IO$_ACCESS,
                              iosb, 0,0,0, &ncb, 0,0,0,0);
  if ((stat != SS$_NORMAL) || (iosb[0] != SS$_NORMAL))
    {
     printf("** Accept error be_chan %d, stat= %d(%x),");
     printf(" iosb[0]= %d(%x) **\n", next_chan, stat,
                                  stat, iosb[0], iosb[0]);
    exit_gracefully();
    }
```

```
    /* Get unit numbers for the channels */
    stat= sys$getdvi(1, be_chan[next_chan],
                                        0,&dvi, iosb, 0,0,0);
    if (stat != SS$_NORMAL)
        {
        printf("** Error getting channel number for BE ");
        printf("%d, stat= %d (%x) **\n",
                                      next_chan, stat,  stat);
        exit_gracefully();
        }
    sys$waitfr(1);
    if (iosb[0] != SS$_NORMAL)
        {
        printf("** Error getting channel number for BE ");
        printf(" %d, iosb[0]= %d(%x) **\n", next_chan,
                                          iosb[0], iosb[0]);
        exit_gracefully();
        }
    be_dev[next_chan] = *dviunit;

      /* increment the available channel pointer */
      next_chan++;

#ifdef EnExFlag
        printf("Exit connect\n");
#endif
}  /* end connect */
```

```
/*     Routine to disconect all of the network links     */
disconnect()
{
 int stat, i;

#ifdef EnExFlag
        printf("Enter Disconnect\n");
#endif

 for (i=1; i < next_chan; i++)
    {
#ifdef pr_flag
        printf("Disconnecting backend %d\n", i);
#endif
    stat= sys$dassgn(be_chan[i]);
    if (stat != SS$_NORMAL)
       printf("** Dassign Error for backend %d. ");
       printf("stat= %d (%x) **\n", i, stat, stat);
    }
#ifdef EnExFlag
        printf("Exit Disconnect\n");
#endif
}


/* SET_HEADER assigns values to the msg header       */
set_header ()
    {
         head.sender = G_PCLB;

         /* set the receiver */
         head.receiver = DM;          /* the default */

    }    /* end set_header */


/**  Close all network connections before aborting     **/
exit_gracefully()
{
   sleep(DELAY);
   disconnect();
   exit();
}
```

```
/*****************************************/
/*                                       */
/*     V A X / V M S   P _ P C L B       */
/*                                       */
/*****************************************/

#include  <stdio.h>     /* Standard I/O definition */
#include  <ssdef.h>     /* VMS return status codes */
#include  <iodef.h>     /* VMS I/O return codes    */
#include  <msgdef.h>    /* DECNET msg definitions  */
#include  "commdata.def"  /* MBDS common defs      */
#include  "msg.def"       /* MBDS msg-type defs    */
#include  "flags.def"     /* MBDS compile flags    */
#include  "beno.dcl"      /* backend num declares  */
#define   MAXBE  16       /* max num of backends   */

char     netbuf[MSGLEN];      /* network buffer     */
char     mbxbuf[MSGLEN];      /* mailbox buffer     */
char     msg[MSGLEN];         /* MBDS-msg buffer    */

short    net_chan;            /* channel to DECNET  */
short    mbx_chan;            /* channel to mailbox */
short    be_chan[MAXBE + 1]; /* chan's to backends */
                        /* backend 0 is controller */

short    NoBackends;          /* the num of backends */
short    next_chan = 0;       /* the next avail chan */

struct sd {
          int  len;
          char *ptr;
          }  /* string descriptors for: */

          /* the network device name     */
          netnam = { 5, "_NET:"  },

          /* the mailbox name            */
          netmbx = { 8, "P_NETMBX" };

struct    msg_hdr   head;
```

```
main ()
   {
   int          i:
   int          StopSys: /* loop flag */
   int          j,k:
   char          NoBEs[NoBElength +1];
   char          BE_Number[NoBElength +1];

#ifdef EnExFlag
 printf("Enter p_pcl_b\n");
#endif

   /* Initialize intra-computer communication */
   initsr (P_PCLB);

   /* set up to use DECNET */
   controller_net_init();

   /* receive a message from the controller */
   receive(&msg[0], &head);

   /* The first message should be of type SetBEno. */
   if ( head.type != SetBEno )
      {
       printf( "Error in PPCL, 1st message must ");
       printf("be of type SetBEno" );
       printf( "  Type = %d\n", head.type ):
       m_prnt(&msg[0], &head);
       exit_gracefully();
       }

   /* get number of backends */
   for( k=0,j=0 ; ( NoBEs[j] = msg[k]) != '\0':
                                          k++, j++ );

   NoBackends = str_to_num( NoBEs ):
   k++;

   /* get backend number */
   for( j=0 ; ( BE_Number[j] = msg[k]) != '\0':
                                          k++, j++ );

   BACKEND_NO = str_to_num( BE_Number );
```

```
backend_net_init(NoBackends, BACKEND_NO);

/*  main portion of program */
StopSys = FALSE;
while ( !StopSys )
  {
   /*recv a msg from a BE process */
   receive(&msg[0], &head);

   if ( head.type == Stop )
      {
      StopSys = TRUE;

      /* send the msg over the network */
      /* backend 0 is the controller   */
      put_message (&msg[0], &head, 0);
      }

   else if ( head.type == DescIds )
     { /* send DescIds to other backends */
     head.receiver = DM;
     for ( i = 1;   i <= NoBackends;   i = i + 1 )
       {
       /* send the DescIds to other backends */
       if ( i != BACKEND_NO )
          {
          put_message (&msg[0], &head, i);
          }
       }/* end for */
     }/* end else if */
   else
     {
     /* send all other messages to controller */
     /* backend 0 is the controller           */
     put_message(msg, &head, 0);
     } /* end else */

  }/* end while */
#ifdef EnExFlag
   printf("Exit p_pcl_b\n");
#endif

} /* end.main */
```

```
/* routine to send a message over the network */

put_message(mptr,hptr,BEnum)

  char    *mptr;
  struct msg_hdr *hptr;
  int     BEnum;

  {
   int    stat,
          msglen,
          func,
          j, k;

   short iosb[4];
   char  sndbuf[MSGLEN + 1],
          intbuf[2] = "\0",    /* null message */
          tmpstr[5];


#ifdef EnExFlag
        printf("Enter put_message\n");
#endif

     hptr -> sender =   P_PCLC;
     hptr -> receiver = G_PCLB;

     k=0;

     /* copy header into message to be sent */

     len_num_to_str(hptr->sender, 3, tmpstr);
     for (j=0; j < 3; j++)
          sndbuf[k++] = tmpstr[j];

     len_num_to_str(hptr->receiver, 3, tmpstr);
     for (j=0; j < 3; j++)
          sndbuf[k++] = tmpstr[j];

     len_num_to_str(hptr->type, 3, tmpstr);
     for (j=0; j < 3; j++)
          sndbuf[k++] = tmpstr[j];
```

```
    /*  copy the message    */
    j=0;
    while ( (k < MSGLEN) &&
         ((sndbuf[k++] = mptr[j++]) != EOMsg) );


    if (k >= MSGLEN)
       printf("***** Value of MSGLEN ");
       printf("should be increased *****\n");

    msglen = k;

    /* send the message */

#ifdef pr_flag
       printf("Calling qio write to backend %d\n",
                                                  BEnum);
       m_prnt(sndbuf, hptr);
#endif
    func = IO$_WRITEVBLK;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb, 0,0,
                                  sndbuf, msglen, 0,0,0,0);
#ifdef pr_flag
       printf("Returned from qio, write\n");
#endif

    if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
      {
      printf("** Write error be_chan %d, stat= %d ");
      printf("(%x), iosb[0]= %d (%x) **\n", BEnum,
                           stat, stat, iosb[0], iosb[0]);
      exit_gracefully();
      }


   /* let receiver know msg is there */
#ifdef pr_flag
       printf("Calling qio, interrupt\n");
#endif
    func = IO$_WRITEVBLK | IO$M_INTERRUPT;
    stat= sys$qiow(0, be_chan[BEnum], func, iosb, 0,0,
                                     intbuf, 1, 0,0,0,0);
```

```
#ifdef pr_flag
        printf("Returned from qio. interrupt n");
#endif

     if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
       {
      printf("** Interrupt error be_chan %d, stat= %d");
      printf(" (%x), iosb[0]= %d(%x) **\n", BEnum,
                              stat, stat, iosb[0], iosb[0]);
       exit_gracefully();
       }

#ifdef EnExFlag
        printf("Exit put_message\n");
#endif

   }
```

```
/* Routine to initialize Decnet to receive connections */

controller_net_init()
{
int    stat, i;
short iosb[4];

#define    NFB$C_DECLNAME 0x15
char       nfb[5] = { NFB$C_DECLNAME, 0,0,0,0 };

struct sd {
          int  len;
          char *ptr;
          }
          objnam = { 5, "PPCLB"  },
          nfb_d  = { 5, nfb      };

char       tmpstr   [5];


#ifdef EnExFlag
       printf("Enter controller_net_init\n");
#endif

  /* create a mailbox */
  stat=sys$crembx(0, &mbx_chan, MSGLEN, MSGMAX,
                                        0, 0, &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Error creating mailbox, ");
     printf("stat= %d (%x) **\n", stat, stat);
     exit_gracefully();
     }

  /* assign channel to the net */
  stat= sys$assign(&netnam, &net_chan, 0, &netmbx);
  if (stat != SS$_NORMAL)
     {
     printf("** Error in assign for netchan, ");
     printf("stat=%d (%x) **\n", stat, stat);
     exit_gracefully();
     }
```

```
    /* declare a network name */
    stat= sys$qiow(0, net_chan, IO$_ACPCONTROL,
                   iosb, 0,0, &nfb_d, &objnam, 0,0,0,0);
    if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
       {
       printf("** Error declaring network name **\n");
       exit_gracefully();
       }

    /* check mailbox for connection requests */
    read_mbx(mbxbuf);

    /* accept the connection */
    connect(mbxbuf);

#ifdef EnExFlag
        printf("Exit controller_net_init\n");
#endif
}
```

```
/* Routine to initialize Decnet links to backends */

backend_net_init(NoBEs, BEnum)

int NoBEs;

{
  int   stat, i;
  short iosb[4];
  char  nodespec[128], tmpstr[5];

  struct sd {
            int  len;
            char *ptr;
            }
            netnam = { 5, "_NET:" },
            ncb    = { 0, nodespec };

#ifdef EnExFlag
       printf("Enter backend_net_init\n");
#endif

  for(i=1; i <= NoBEs; i++)
    {
    if (i != BEnum )
       {
       /* assign a channel to the net */
       stat= sys$assign(&netnam, &be_chan[i], 0, 0);
       if (stat != SS$_NORMAL)
          {
          printf("** Error in assign be_chan %d, ");
          printf("stat= %d (%x) **\n", 1, stat, stat);
          exit_gracefully();
          }
        }  /* end if i != BEnum */
     }     /* end for */
```

```
   /* Establish logocal link */
   for (i=1; i <= NoBEs; i++)
        {
        if (i != BEnum )
           {
           /* build network connect block */
           strcpy(nodespec, "CSMV");
           num_to_str(i, tmpstr);
           strcat (nodespec, tmpstr);
           strcat (nodespec, "::\"0=GPCLB\"");

#ifdef pr_flag
  printf("backend %d nodespec. ");
  printf("\"%s\"\n", i, nodespec);
#endif

        ncb.len= strlen(nodespec);

        /*  Request the connection */
        stat= sys$qiow(0, be_chan[i], IO$_ACCESS,
                              iosb, 0,0,0, &ncb, 0,0,0,0);
        if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
          {
          printf("* Access error be_chan %d. ");
          printf("stat= %d(%x), iosb[0]= %d(%x) *\n",
                          i, stat, stat, iosb[0], iosb[0]);
          exit_gracefully();
          }
        }  /* end if i != BEnum  */
     }     /* end for i */

#ifdef EnExFlag
        printf("Exit backend_net_init\n");
#endif
}
```

```
/*  Routine to check the mailbox for message notices */

read_mbx(buf)

char *buf;
{
short   connect_rec;
short   stat;
short   iosb[4];

#ifdef EnExFlag
        printf("Enter read_mbx\n");
#endif

    connect_rec = FALSE;

    while ( ! connect_rec )
      {
#ifdef pr_flag
        printf("Calling qio, read mailbox\n");
#endif
      stat= sys$qiow(0, mbx_chan, IO$_READVBLK,
                        iosb, 0,0, buf, MSGLEN, 0,0,0,0);
#ifdef pr_flag
        printf("Returned from qio \n");
#endif
      if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
        {
        printf("** Mailbox read error, stat= %d (%x),");
        printf(" iosb[0]= %d (%x) **\n", stat, stat,
                                        iosb[0], iosb[0]);
        exit_gracefully();
        }

      switch (buf[0])
        {
        case MSG$_CONNECT:
                        {
                        connect_rec = TRUE;
                        break;
                        }
```

```
        case MSG$_CONFIRM:
        case MSG$_INTMSG : break;

        default:        {
                        printf("** Network error, ");
                        printf("mbxbuf[0]= %d (%x) **\n",
                                        buf[0], buf[0]);
                        exit_gracefully();
                        }
        }  /* end switch */
      } /* end while */

#ifdef EnExFlag
        printf("Exit read_mbx\n");
#endif

}  /* end read_mbx */
```

```
/* routine to accept a network connection request */

connect(buf)

char *buf;
{
short       i,
            offset,
            stat,
            iosb[4];

char        nodespec[128];

struct sd {
            int  len;
            char *ptr:
            }
            ncb     = { 0, nodespec };

#ifdef EnExFlag
        printf("Enter connect\n");
#endif

  /* Extract network connect block from mailbox buf */

  offset  = buf[4] + 5:    /* point to ncb length     */
  ncb.len = buf[offset];   /* put the len in our ncb  */
  offset++;                /* point past ncb length   */
  for (i=0; i < ncb.len; i++)      /* get the ncb     */
      nodespec[i] = buf [i + offset]:
  nodespec[i]= '\0';

#ifdef pr_flag
        printf("** nodespec= %s **\n", nodespec);
        printf("** next_chan= %d **\n", next_chan);
#endif
```

```
    /* Assign controller channel to the net */
    stat= sys$assign(&netnam, &be_chan[0], 0, &netmbx);
    if (stat != SS$_NORMAL)
        {
        printf("** Assign error be_chan %d, ");
        printf("stat= %d (%x)  **\n", next_chan,
                                          stat, stat);
        exit_gracefully();
        }

    /* accept the connection */
    stat= sys$qiow(0, be_chan[0], IO$_ACCESS,
                          iosb, 0,0,0, &ncb, 0,0,0,0);
    if ((stat != SS$_NORMAL)||(iosb[0] != SS$_NORMAL))
      {
       printf("** Accept error be_chan %d, ");
       printf("stat= %d(%x), iosb[0]= %d(%x) **\n",
              next_chan, stat, stat, iosb[0], iosb[0]);
      exit_gracefully();
      }

#ifdef EnExFlag
        printf("Exit connect\n");
#endif
}  /* end connect */
```

```
/* routine to disconnect all network links */

disconnect()
{
 int stat, i;

#ifdef EnExFlag
        printf("Enter Disconnect\n");
#endif

 for (i=1; i <= NoBackends; i++)
    {
#ifdef pr_flag
        printf("Disconnecting backend %d\n", i);
#endif
    stat= sys$dassgn(be_chan[i]);
    if (stat != SS$_NORMAL)
     printf("** Deassign Error for backend %d. ");
     printf("stat= %d (%x) **\n",i,stat, stat);
    }
#ifdef EnExFlag
        printf("Exit Disconnect\n");
#endif

}
```

```
/* Routine to close network connections then abort */
exit_gracefully()
{
    sleep(DELAY);
    disconnect();
    exit();
}
```

LIST OF REFERENCES

1. Boyne, R. D., Hsiao, D. K., Kerr, D. S., and Orooji, A., A message-oriented implementation of a multi-backend database system (MBDS). *Database Machines*, Leilich and Missikoff (eds.), Springer-Verlag, (1983).

2. Demurjian, S. A., Hsiao, D. K., and Menon, M. J., A multi-backend database system for performance gains, capacity growth and hardware upgrade. *Proceedings of the Second International Conference on Data Engineering*, (1986).

3. Kerr, D.S., Orooji, A., Shi, Z. and Strawser, P. R., The implementation of a multi-backend database system (MBDS): part I - software engineering strategies and efforts towards a prototype MBDS. *Advanced Database Machine Architecture*, Hsiao (ed.), Prentice-Hall, (1983).

4. He, X., Higashida, M., Hsiao, D. K., Kerr, D. S., Orooji, A., Shi, Z. and Strawser, P. R., The implementation of a multi-backend database system (MBDS): part II - the first prototype MBDS and the software engineering experience. *Advanced Database Machine Architecture*, Hsiao (ed.), Prentice-Hall, (1983).

5. Wong, A., Towards highly portable database systems: issues and solutions. Master's Thesis, Naval Postgraduate School, Monterey, California, (to be published).

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
4.	Curriculum Officer, Code 37 Computer Technology Programs Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
7.	LT Bruce D. Silberman 1720 Seaton Drive Virginia Beach, Va 23462	6

219332

Thesis
S494155
c.1

Silberman
Software portability: a case of the multi-backended database system.